Programación

Programación Orientada a Objetos (II)

Unidad 9

Jesús Alberto Martínez
versión 0.2

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

 Importante	 Atención	 Interesante

Unidad 9. Programación Orienta a Objetos (II)

1 Introducción

Los conceptos de la programación orientada a objetos van mucho más hallá de lo visto hasta ahora clases, objetos, métodos, atributos, encapsulación, visibilidad, etc.

En este tema trataremos el resto de conceptos que nos quedan por ver, herencia, polimorfismo, interfaces y clases abstractas.

Pero comenzaremos viendo algo que no está relacionado con la orientación a objetos, si no que es una parte del lenguaje de programación Java, las Enumeraciones.

2 Enumeraciones

Imaginemos que tenemos que modelar una clase para Pokemon, entre otros atributos, podríamos tener uno que nos indicara el tipo de Pokemon que es. Éste podría ser PLANTA, TIERRA, BICHO, etc.

Hasta ahora podemos modelarlos como constantes de clase, y asignarles del tipo entero, o también podríamos asignarle de tipo String.

```
public static final TIPO_PLANTA=1;
public static final TIPO_TIERRA=2;
```

Así podemos tener un constructor simple donde le indicáramos el nombre del Pokemon y su tipo:

```
public Pokemon(String nombre, int tipo)
```

Y donde la forma correcta de usarlo sería

```
new Pokemon("Mew",Pokemon.TIPO_PSIQUICO);
```

Hasta aquí todo bien, pero, ¿Quien impide usar directamente los enteros? No podemos evitar que se pueda llamar de esta forma.

```
new Pokemon("Mew",4);
```

Aunque dentro si podriamos hacer las comprobaciones necesarias para al menos asegurarnos de que el número sea correcto.

Para solucionar este tipo de problemas surgen las Enumeraciones, tratadas como un tipo o clase, donde se definen una lista de posibles valores que puede contener. Nos facilita la creación de nuevos tipos de datos, bien definidos y precisos.

Se definen mediante la palabra clave enum seguido del nombre que le damos a la enumeración y a continuación la lista de valores, entre llaves y separados por comas. No son cadenas de caracteres.

```
enum TipoPokemon {TIERRA, PLANTA, ELECTRICO, PSIQUICO};
```

y cuando definimos al atributo tipo de Pokemon será

```
private TipoPokemon tipo;
```

en el constructor también usaríamos la enumeración

```
public Pokemon(String nombre, TipoPokemon tipo)
```

y cuando creamos un nuevo Pokemon obligatoriamente debemos usar uno de los tipos definidos en la enumeración

```
new Pokemon("Mew", TipoPokemon.PSIQUICO);
```

Ahora está perfectamente definido un nuevo tipo de datos, TipoPokemon. Internamente, como todo en Java, se trata de una nueva clase.

Podemos usar el método toString() para obtener el tipo en forma de cadena de caracteres.

```
System.out.println(pokemon.getTipo());
```

Y puedo comprobar la igualdad de tipos con el símbolo == por ejemplo dentro de una estructura condicional.

```
public ArrayList<Pokemon> listaByTipo(TipoPokemon tipo){
    ArrayList<Pokemon> lista=new ArrayList<>();
    for (Pokemon pokemon : mochila) {
        if (pokemon.getTipo()==tipo){
            System.out.println(pokemon);
            lista.add(pokemon);
        }
    }
    return lista;
}
```

3 Herencia

3.1 Introducción

La herencia es una de las capacidades más importantes y distintivas de la POO. Consiste en derivar o extender una clase nueva a partir de otra ya existente de forma que la clase nueva hereda todos los atributos y métodos de la clase ya existente.

Otra posible definición, “La herencia es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente los métodos y atributos entre clases y sus clases sucesoras”

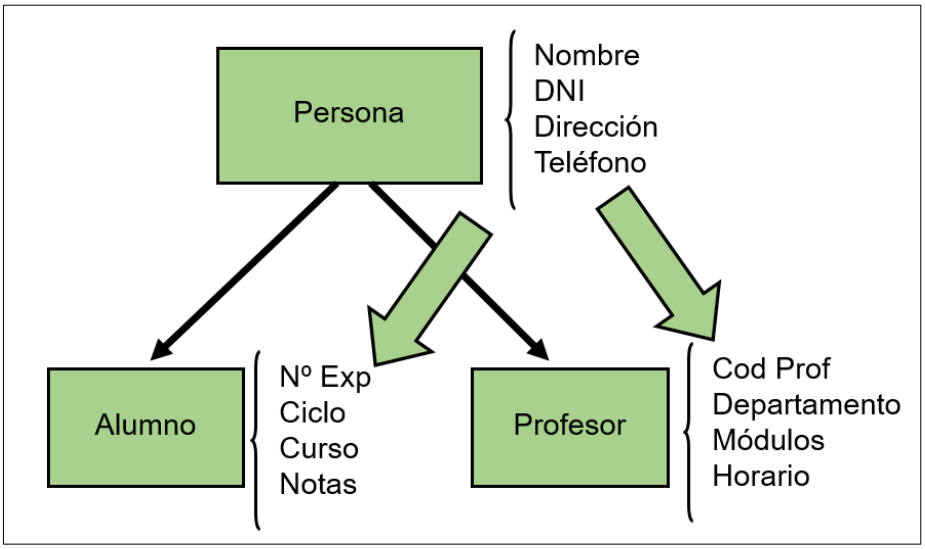
A la clase ya existente se la denomina superclase, clase base o clase padre. A la nueva clase se la denomina subclase, clase derivada o clase hija.

> Cuando derivamos (o extendemos) una nueva clase, ésta hereda todos los datos y métodos miembro de la clase existente.
> Todas las clases heredan de otra, si no se pone nada, hereda de la superclase Object. Una clase puede tener cualquier cantidad de subclases, pero una clase sólo tiene una clase padre. En otros lenguajes de programación se pueden heredar de carias clases, herencia múltiple.

Por ejemplo, si tenemos un programa que va a trabajar con alumnos y profesores, éstos van a tener atributos comunes como el nombre, dni, dirección o teléfono. Pero cada uno de ellos tendrán atributos específicos que no tengan los otros. Por ejemplo los alumnos tendrán el número de expediente, el ciclo y curso que cursan y sus notas; por su parte los profesores tendrán el código de profesor, el departamento al que pertenecen, los módulos que imparten y su horario.

Por lo tanto, en este caso lo mejor es declarar una clase Persona con los atributos comunes (Nombre, DNI, Dirección, Teléfono) y dos subclases Alumno y Profesor que hereden de Persona (además de tener sus propios atributos).

Es importante recalcar que Alumno y Profesor también heredarán todos los métodos de Persona.

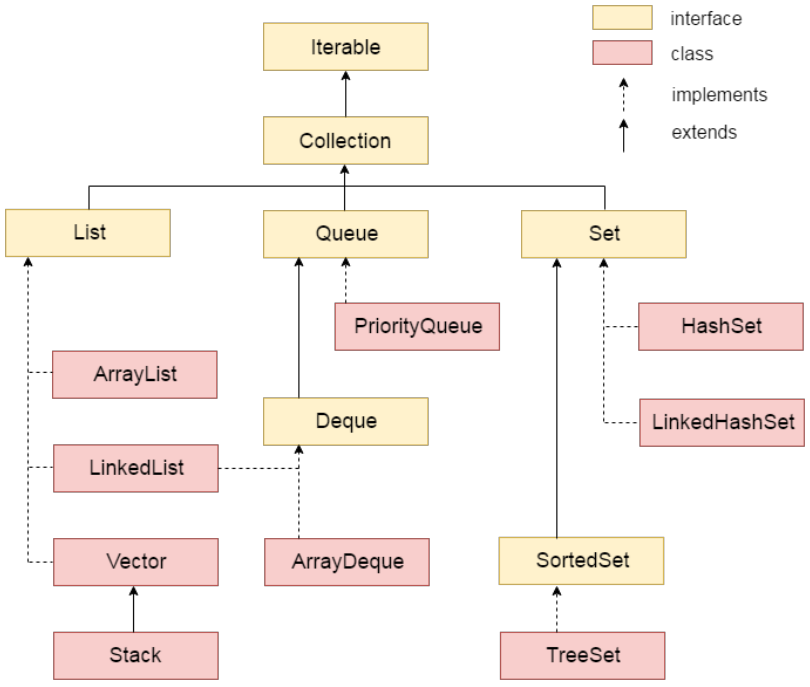


En Java se utiliza la palabra reservada extends para indicar herencia:

```
public class Alumno extends Persona {
      ...
}

public class Profesor extends Persona {
      ...
}
```

Toda la POO se basa en la herencia. Ejemplo de jerarquía de clases, Collection Framework



3.2 Constructores de clases derivadas

El constructor de una clase derivada debe encargarse de construir los atributos que estén definidos en la clase base además de sus propios atributos.

Dentro del constructor de la clase derivada, para llamar al constructor de la clase base se debe utilizar el método reservado super() pasándole como argumento los parámetros que necesite.

Si no se llama explícitamente al constructor de la clase base mediante super() el compilador llamará automáticamente al constructor por defecto de la clase base. Si no tiene constructor por defecto el compilador generará un error.

3.3 Métodos heredados y sobreescritos

Hemos visto que una subclase hereda los atributos y métodos de la superclase; además, se pueden incluir nuevos atributos y nuevos métodos.

Por otro lado, puede ocurrir que alguno de los métodos que existen en la superclase no nos sirvan en la subclase (tal y como están programados) y necesitemos adecuarlos a las características de la subclase. Esto puede hacerse mediante la sobreescritura de métodos:

Un método está sobreescrito o reimplementado cuando se programa de nuevo en la clase derivada.

La "firma" del método debe ser la misma a la del método que estamos sobreescribiendo. Entendemos firma de un método a su nombre, los parámetros que recibe y el valor que devueve. Además el nivel de acceso no puede ser más restrictivo que la del padre.

Algo que también debemos mencionar es que podemos identificar un método sobrescrito cuando tiene la anotación @Override, sin embargo, no es obligatorio ponerlo, pero si es recomendable (pues de esta manera el compilador reconoce que este método está sobrescribiendo un método de una superclase de ésta, ayudando a que, si nos equivocamos al construirlo, el compilador nos avisaría. Adicionalmente si tenemos la anotación inmediatamente podremos saber que se está aplicando el concepto, algo muy útil cuando trabajamos con código de otras personas.

Las anotaciones son metadatos que se pueden asociar a clases, miembros, métodos o parámetros. No cambian las acciones de un programa, pero dan información sobre el elemento que tiene la anotación y permiten definir cómo queremos que sea tratado por distintas herramientas (en la compilación, documentación, ejecución, etc.) Comienzan siempre por @.
Algunas de las más comunes son @Override, @SuprressWarnings, @Deprecated, etc.

Por ejemplo el método mostrarPersona() de la clase Persona lo necesitaríamos sobreescribir en las clases Alumno y Profesor para mostrar también los nuevos atributos.

El método sobreescrito en la clase derivada podría reutilizar el método de la clase base, si es necesario, y a continuación imprimir los nuevos atributos. En Java podemos acceder a método definidos en la clase base mediante super.metodo().

El método mostrarPersona sobreescrito en las clases derivadas podría ser:

```
   super.mostrarPersona(); // Llamada al método de la clase base
   System.out.println(…); // Imprimimos los atributos exclusivos de la
clase derivada
```

3.4 Clases y métodos final

- Una clase ***final*** no puede ser heredada.
- Un método *final* no puede ser sobreescrito por las subclases.

3.5 Acceso a miembros derivados

Aunque una subclase incluye todos los miembros de su superclase, no podrá acceder a aquellos que hayan sido declarados como private.

Si en el ejemplo 3 intentásemos acceder desde las clases derivadas a los atributos de la clase Persona (que son privados) obtendríamos un error de compilación.

También podemos declarar los atributos como protected. De esta forma podrán ser accedidos desde las clases heredadas, (nunca desde otras clases).

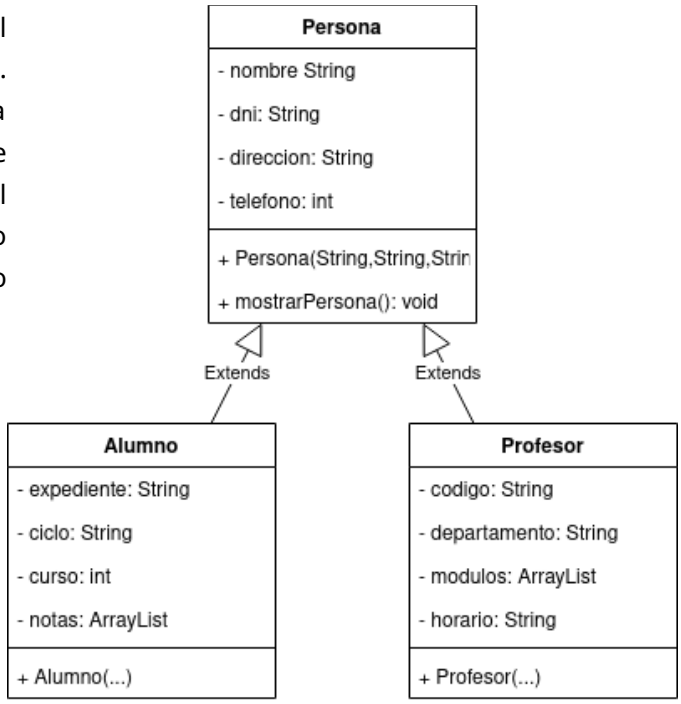
- Los atributos declarados como *protected* son públicos para las clases heredadas y privados para las demás clases.

3.6 Ejemplo 3

En este ejemplo vamos a crear las clase Persona y sus clases heredadas: Alumno y Profesor.

En la clase Persona crearemos el constructor, un método para mostrar los atributos y los getters y setters. Las clases Alumno y Profesor heredarán de la clase Persona (utilizando la palabra reservada extends) y cada una tendrá sus propios atributos, un constructor que llamará también al constructor de la clase Persona (utilizando el método super()), un método para mostrar sus atributos, que también llamará al método de Persona y los getters y setters.

Es interesante ver cómo se ha sobreescrito el método mostrarPersona() en las clases heredadas. El método se llama igual y hace uso de la palabra reservada super para llamar al método de mostrarPersona() de Persona. En la llamada del main tanto el objeto a (Alumno) como el objeto profe (Profesor) pueden hacer uso del método mostrarPersona().



Persona

```java
public class Persona {
    private String nombre;
    private String dni;
    private String direccion;
    private int telefono;

    // No se ponen los comentarios javaDoc
    public Persona(String nombre, String dni, String direccion, int telefono)
{
        this.nombre = nombre;
        this.dni = dni;
        this.direccion = direccion;
        this.telefono = telefono;
    }

    public void mostrarPersona() {
        System.out.println("Nombre: " + this.nombre);
        System.out.println("DNI: " + this.dni);
        System.out.println("Dirección: " + this.direccion);
        System.out.println("Telefono: " + this.telefono);
    }

    public String getNombre() {
```

```
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getDni() {
        return dni;
    }

    public void setDni(String dni) {
        this.dni = dni;
    }

    public String getDireccion() {
        return direccion;
    }

    public void setDireccion(String direccion) {
        this.direccion = direccion;
    }

    public int getTelefono() {
        return telefono;
    }

    public void setTelefono(int telefono) {
        this.telefono = telefono;
    }
}
```

Alumno

(hereda de Persona)

```
import java.util.ArrayList;

public class Alumno extends Persona {

    private int expediente;
    private String ciclo;
    private int curso;
    private ArrayList notas;
    public Alumno(String nombre, String dni, String direccion, int telefono,
int expediente, String ciclo, int curso,
```

```
        ArrayList notas) {
        // Llamamos al constructor del padre
        super(nombre, dni, direccion, telefono);
        this.expediente = expediente;
        this.ciclo = ciclo;
        this.curso = curso;
        this.notas = notas;
    }
    @Override
    public void mostrarPersona() {
        // Llamamos al método del padre
        super.mostrarPersona();
        System.out.println("Núm. expediente: " + this.expediente);
        System.out.println("Ciclo: " + this.ciclo);
        System.out.println("Curso: " + this.curso);
        System.out.println("Notas:");
        for (Object nota: notas) {
            System.out.println("\nNota: " + nota);
        }
    }
    public int getExpediente() {
        return expediente;
    }
    public void setExpediente(int expediente) {
        this.expediente = expediente;
    }
    public String getCiclo() {
        return ciclo;
    }
    public void setCiclo(String ciclo) {
        this.ciclo = ciclo;
    }
    public int getCurso() {
        return curso;
    }
    public void setCurso(int curso) {
        this.curso = curso;
    }
    public ArrayList getNotas() {
        return notas;
    }
    public void setNotas(ArrayList notas) {
        this.notas = notas;
    }
}
```

Profesor

(hereda de Persona)

```java
import java.util.ArrayList;
import java.util.Iterator;

public class Profesor extends Persona {

    private int codigo;
    private String departamento;
    private ArrayList modulos;
    private String horario;

    // Al constructor le pasamos los atributos de Persona y de Profesor
    public Profesor(String nombre, String dni, String direccion, int
telefono, int codigo, String departamento,
            ArrayList modulos, String horario) {
        // LLamamos al constructor del padre
        super(nombre, dni, direccion, telefono);
        this.codigo = codigo;
        this.departamento = departamento;
        this.modulos = modulos;
        this.horario = horario;
    }

    @Override
    public void mostrarPersona() {
        // Llamamos al método del padre
        super.mostrarPersona();
        System.out.println("Código: " + this.codigo);
        System.out.println("Departamento " + this.departamento);
        System.out.println("Modulos: ");
        for (Iterator it = modulos.iterator(); it.hasNext();) {
            System.out.println("\nMódulo: " + it.next());
        }
        System.out.println("Horario:");
    }

    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }
```

```
    public String getDepartamento() {
        return departamento;
    }

    public void setDepartamento(String departamento) {
        this.departamento = departamento;
    }

    public ArrayList getModulos() {
        return modulos;
    }

    public void setModulos(ArrayList modulos) {
        this.modulos = modulos;
    }

    public String getHorario() {
        return horario;
    }

    public void setHorario(String horario) {
        this.horario = horario;
    }
}
```

Programa Principal

```
import java.util.ArrayList;

public class Herencia {

    public static void main(String[] args) {
        // Creamos una Persona, con su constructor
        Persona p = new Persona("Pepe", "12345678T", "C/ del viento",
66666666);

        System.out.println("----------- Mostramos una persona");
        p.mostrarPersona();

        // Creamos un Alumno
        ArrayList notas = new ArrayList();
        notas.add(6);
        notas.add(8);
        notas.add(9);
```

```
        // nombre,dni,dirección,teléfono,expediente,ciclo,curso,notas
        Alumno a = new Alumno("Maria", "11111111A", "C/ del mar", 111111111,
1, "DAM", 1, notas);
        System.out.println("----------- Mostramos un alumno");
        a.mostrarPersona();

        // Creamos un Profesor
        ArrayList modulos = new ArrayList();
        modulos.add("Programación");
        modulos.add("Entornos de desarrollo");

        // nombre,dni,dirección,teléfono,código,departamento,módulos,horario
        Profesor prof = new Profesor("Juan", "22222222B", "C/ del fin",
222222222, 2, "Teología", modulos, "Nocturno");
        System.out.println("----------- Mostramos un profesor");
        prof.mostrarPersona();
    }
}
```

Salida:

```
----------- Mostramos una persona
Nombre: Pepe
DNI: 12345678T
Dirección: C/ del viento
Telefono: 66666666
----------- Mostramos un alumno
Nombre: Maria
DNI: 11111111A
Dirección: C/ del mar
Telefono: 111111111
Núm. expediente: 1
Ciclo: DAM
Curso: 1
Notas:
        Nota: 6
        Nota: 8
        Nota: 9
----------- Mostramos un profesor
Nombre: Juan
DNI: 22222222B
Dirección: C/ del fin
Telefono: 222222222
Código: 2
Departamento Teología
Modulos:
        Módulo: Programación
        Módulo: Entornos de desarrollo
Horario:Nocturno
```

3.7 Métodos comunes

Como hemos mencionado anteriormente, todas las clases derivan de una u otra forma de la superclase Object. Esta clase define una serie de métodos, éstos se puede usar así como están pero podemos redefinirlos.

getClass

Devuelve la clase en tiempo de ejecución del objeto sobre el que se llama. Si lo imprimimos, muestra una referencia que incluye el nombre de la clase y el paquete en el que se encuentra, pero podemos obtener más información, como el nombre concreto de la clase (sin el paquete):

```
obj.getClass().getSimpleName();
```

operador instanceof

Es parecido al método anterior, pero es un operados que nos permite preguntar el tipo (clase) de la variable. Ejemplo:

```
if (p instanceof Alumno)
      System.out.println (((Alumno)p).getExpediente());
```

instanceof devuelve un valor boolean indicando si la instancia pertenece a la clase o no. Hay que resaltar que también devuelve true para todas las clases ancestro (padre, abuelo, etc) de la clase a la que pertenece.

En nuestro ejemplo, si p es un Alumno, devolvería verdadero para Alumno, pero también para Persona. Para Profesor devolvería false.

toString

El método toString(), como su propio nombre indica, se utiliza para convertir a String (es decir, a una cadena de texto) cualquier objeto. Este método está definido para la clase Object, y como todos los objetos heredan de dicha clase, todos ellos tienen acceso a este método.

Este método está sobrescrito para muchas clases como por ejemplo LocalDateTime o ArrayList mostrando el contenido del objeto de una forma "comprensible" para el usuario. Así podemos hacer:

```
System.out.println(LocalDate.now().toString());
```

De hecho, System.out.println usa por defecto el método toString() para mostrar el contenido de su parámetro, siempre que esté definido, así pues, el ejemplo anterior podría escribirse así:

```
System.out.println(LocalDate.now());
```

Deberíamos, entonces, redefinir el método toString () para las clases que creemos, así facilitaremos su representación. Si no lo definimos, se mostraría una representación que incluye el nombre de su clase y una referencia de su ubicación en memoria.

> ArrayList tiene definido el método toString() por lo que podemos hacer System.out.println(miArraylist) pero no así en el caso de Array, que debemos emplear el método estático Arrays.toString para convertirlo previamente en en String y poder hacer System.out.println(Arrays.toString(miArray))

equals

Es un caso similar al toString() ya que está definido para la clase padre de todos los objetos Object, sobrescrito en otras clases como String y que debemos sobrescribir para nuestras clases.

Ya hemos utilizado el método equals () previamente, y sabemos que es la forma en que debemos comparar objetos (los objetos no se pueden comparar utilizando el operador ==). El método equals está sobrescrito para la mayoría de las clases. Por ello, podemos usarlo directamente para comparar instancias de String o LocalDate, por ejemplo. Pero ¿qué ocurre en las clases que creamos nosotros?

> Si no sobrescribimos equals() solo devolverá true en caso de que las dos referencias comparadas compartan la misma ubicación de memoria, es decir apunten a la misma instancia de objeto.

Para que equals funcione con nuestros objetos tendremos que sobreescribir el método y devolver verdadero o falso realizando todas las comprobaciones necesarias. Los entornos de desarrollo nos facilitarán la sobreescritura de este método.

Por ejemplo, si queremos comparar dos objetos de la clase Persona, lo que queremos comprobar es si todos sus atributos son iguales. Tendremos que sobreescribir el método equals para que devuelva verdadero o falso.

```
@Override
public boolean equals(Object o) {

    Persona persona = (Persona) o;

    if (telefono ≠ persona.telefono) return false;
    if (!nombre.equals(persona.nombre)) return false;
    if (!dni.equals(persona.dni)) return false;
    return direccion.equals(persona.direccion);
}
```

hashCode

Este método se utiliza en vez de equals en determinadas ocasiones para determinar si un elemento de una colección de tipo hash es igual a otra instancia, por lo que siempre que sobrescribamos equals() para una determinada clase, deberíamos sobrescribir también hashCode(). Los dos métodos tienen que ir a la par, de forma que si para dos instancias, equals() devuelve true, hashCode() debe devolver el mismo número, pero lo veremos en detalle en el capítulo dedicado a Colecciones.

4 Polimorfismo y ligadura dinámica

La sobreescritura de métodos constituye la base de uno de los conceptos más potentes de Java: la selección dinámica de métodos, que es un mecanismo mediante el cual la llamada a un método sobreescrito se resuelve en tiempo de ejecución y no durante la compilación. La selección dinámica de métodos es importante porque permite implementar el polimorfismo durante el tiempo de ejecución. Una variable que referencia a una superclase se puede referir a un objeto de una subclase. Java se basa en esto para resolver llamadas a métodos sobreescritos en el tiempo de ejecución.

Lo que determina la versión del método que será ejecutado es el tipo de objeto al que se hace referencia y no el tipo de variable de referencia.

El polimorfismo es fundamental en la programación orientada a objetos porque permite que una clase general especifique métodos que serán comunes a todas las clases que se deriven de esa misma clase. De esta manera las subclases podrán definir la implementación de alguno o de todos esos métodos.

La superclase proporciona todos los elementos que una subclase puede usar directamente. También define aquellos métodos que las subclases que se deriven de ella deben implementar por sí mismas. De esta manera, combinando la herencia y la sobreescritura de métodos, una superclase puede definir la forma general de los métodos que se usarán en todas sus subclases

4.1 Ejemplo 4

Vamos probar un ejemplo sencillo pero que resume todo lo importante del polimorfismo.

Vamos a crear la clase Madre con un método llamame(). A continuación crearemos dos clases derivadas de ésta: Hija1 e Hija2, sobreescribiendo el método llamame(). En el main crearemos un objeto de cada clase y los asignaremos a una variable de tipo Madre (llamada madre2) con la que llamaremos al método llamame() de los tres objetos.

Es importante observar que la variable Madre madre2 se puede asignar a objetos de clase Hija1 e Hija2. Esto es posible porque Hija1 e Hija2 también son de tipo Madre (debido a la herencia).

También es importante ver que la variable Madre madre2 llamará al método llamame() de la clase del objeto al que hace referencia (debido al polimorfismo).

```
10  class Madre {
11      void llamame(){
12          System.out.println("Estoy en la clase Madre");
13      }
14  }
15
16  class Hija1 extends Madre {
17      void llamame(){
18          System.out.println("Estoy en la subclase Hija1");
19      }
20  }
21
22  class Hija2 extends Madre {
23      void llamame(){
24          System.out.println("Estoy en la subclase Hija2");
25      }
26  }
27
28  class Ejemplo {
29      public static void main(String args[]){
30          // Creamos un objeto de cada clase
31          Madre madre = new Madre();
32          Hija1 h1 = new Hija1();
33          Hija2 h2 = new Hija2();
34
35          // Declaramos otra variable de tipo Madre
36          Madre madre2;
37
38          // Asignamos a madre2 el objeto madre
39          madre2 = madre;
40          madre2.llamame();
41
42          // Asignamos a madre2 el objeto h1 (Hija1)
43          madre2 = h1;
44          madre2.llamame();
45
46          // Asignamos a madre2 el objeto h2 (Hija2)
47          madre2 = h2;
48          madre2.llamame();
49      }
50  }
```

Obsérvese las llamadas madre2.llamame() de las líneas 36 en adelante:

- En el primero se invoca al método llamame() de la clase Madre porque 'madre2' hace referencia a un objeto de la clase Madre.
- En el segundo se invoca al método llamame() de la clase Hija1 porque ahora 'madre2' hace referencia a un objeto de la clase Hija1.
- En el tercero se invoca al método llamame() de la clase Hija2 porque ahora 'madre2' hace referencia a un objeto de la clase Hija2.

```
run:
Estoy en la clase Madre
Estoy en la subclase Hija1
Estoy en la subclase Hija2
BUILD SUCCESSFUL (total time: 0 seconds)
```

Si tomamos como referencia el Ejemplo 3 de Persona, Alumno y Profesor, podríamos hacer esto:

```
p.mostrarPersona();  // mostraría la versión de Persona
p=a;
p.mostrarPersona();  // ahora, p es una Alumno, mostraría versión Alumno
p=prof;
p.mostrarPersona();  // ahora p es un Profesor, mostraría versión Profesor
```

Si quisiéramos tener un ArrayList con Profesores y Alumnos la forma adecuada de definirlo sería a través de su clase padre, Persona.

```
ArrayList<Persona> componentes1DAM = new ArrayList<>();
```

Así puede contener tanto objetos de Persona, como objetos de cualquiera de sus clases hijas, ya que sus clases hijas SON Personas.

Ahora si queremos recorrerlo y mostrar los datos de cada uno de ellos, usaremos

```
for (Persona elemento: componentes1DAM){
  elemento.mostrarPersona();
}
```

Cuando un objeto se evalúe, se llamará al método mostrarPersona de cada uno de ellos, si es un Profesor llamará al mostarPersona de Profesor, y si es Alumno, mostrará la información que devuelve el método de Alumno.

Pero imaginaros que queremos mostrar el número de expediente de los alumnos, pero están mezclados, no podemos hacer elemento.getExpediente(), daría error, ya que Persona, que es estáticamente en compilación lo que Java supone que es, no tiene getExpediente. En tiempo de ejecución, puede cambiar de forma (polimorfismo) y ser un Alumno, pero en tiempo de compilación no puedes saberlo.

Se puede solucionar comprobando la clase a la que pertenece en tiempo de ejecución y realizando un cast a la clase adecuada. Por ejemplo

```
for (Persona elemento: componentes1DAM){
  if (elemento instanceof Alumno){
    Alumno miAlumno=(Alumno)elemento;
    System.out.println("Expediente: "+miAlumno.getExpediente();
  }
}
```

5 Clases Abstractas

Una clase abstracta es una clase que declara la existencia de algunos métodos pero no su implementación (es decir, contiene la cabecera del método pero no su código). Los métodos sin implementar son métodos abstractos.

Una clase abstracta puede contener tanto métodos abstractos (sin implementar) como no abstractos (implementados). Pero al menos uno debe ser abstracto.

Para declarar una clase o método como abstracto se utiliza el modificador abstract.

> Una clase abstracta no se puede instanciar, pero sí heredar. Las subclases tendrán que implementar obligatoriamente el código de los métodos abstractos (a no ser que también se declaren como abstractas).

Las clases abstractas son útiles cuando necesitemos definir una forma generalizada de clase que será compartida por las subclases, dejando parte del código en la clase abstracta (métodos "normales") y delegando otra parte en las subclases (métodos abstractos).

> No pueden declararse constructores o métodos estáticos abstractos.

La finalidad principal de una clase abstracta es crear clases heredadas a partir de ella. Por ello, en la práctica es obligatorio aplicar herencia (si no, la clase abstracta no sirve para nada). El caso contrario es una clase *final*, que no puede heredarse como ya hemos visto. Por lo tanto una clase no puede ser *abstract* y *final* al mismo tiempo.

Por ejemplo, esta clase abstracta Principal tienes dos métodos: uno concreto y otro abstracto.

```
public abstract class Principal {
       // Método concreto con implementación
       public void metodoConcreto() { ... }
    // Método abstracto sin implementación
    public abstract void metodoAbstracto();
}
```

Esta subclase hereda de Principal ambos métodos, pero está obligada a implementar el código del método abstracto.

```
class Secundaria extends Principal {
       // Implementación concreta
       public void metodoAbstracto() { ... }
}
```

6 Interfaces

Una interfaz es una declaración de atributos y métodos sin implementación (sin definir el código de los métodos). Se utilizan para definir el conjunto mínimo de atributos y métodos de las clases que implementen dicha interfaz. En cierto modo, es parecido a una clase abstracta con todos sus miembros abstractos.

Si una clase es una plantilla para crear objetos, una interfaz es una plantilla para crear clases.

Un interfaz es una declaración de atributos y métodos sin implementación.

Mediante la construcción de un interfaz, el programador pretende especificar qué caracteriza a una colección de objetos e, igualmente, especificar qué comportamiento deben reunir los objetos que quieran entrar dentro de sea categoría o colección.

En una interfaz también se pueden declarar constantes que definen el comportamiento que deben soportar los objetos que quieran implementar esa interfaz.

La sintaxis típica de una interfaz es la siguiente:

```
public interface Nombre {
// Declaración de atributos y métodos (sin definir código)
}
```

La implementación (herencia) de una interfaz no podemos decir que evite la duplicidad de código o que favorezca la reutilización de código puesto que realmente no proveen código.

En cambio sí podemos decir que reúne las otras dos ventajas de la herencia: favorecer el mantenimiento y la extensión de las aplicaciones. ¿Por qué? Porque al definir interfaces permitimos la existencia de variables polimórficas y la invocación polimórfica de métodos.

Un aspecto fundamental de las interfaces en Java es separar la especificación de una clase (qué hace) de la implementación (cómo lo hace). Esto se ha comprobado que da lugar a programas más robustos y con menos errores.

Es importante tener en cuenta que:

- Una interfaz no se puede instanciar en objetos, solo sirve para implementar clases.
- Una clase puede implementar varias interfaces (separadas por comas).
- Una clase que implementa una interfaz debe de proporcionar implementación para todos y cada uno de los métodos definidos en la interfaz.

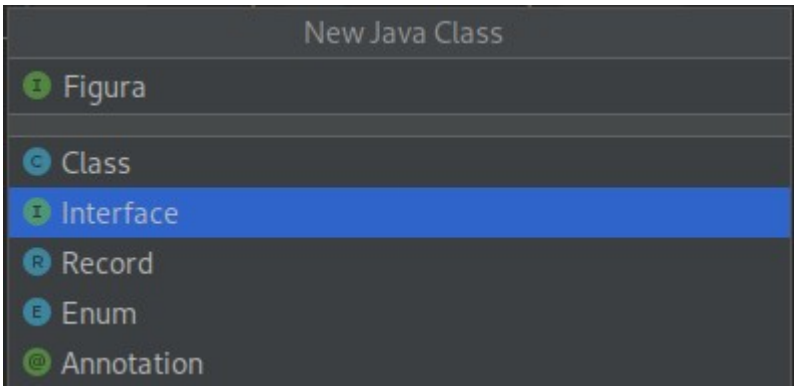
- Las clases que implementan una interfaz que tiene definidas constantes pueden usarlas en cualquier parte del código de la clase, simplemente indicando su nombre.

Si por ejemplo la clase Círculo implementa la interfaz Figura la sintaxis sería:

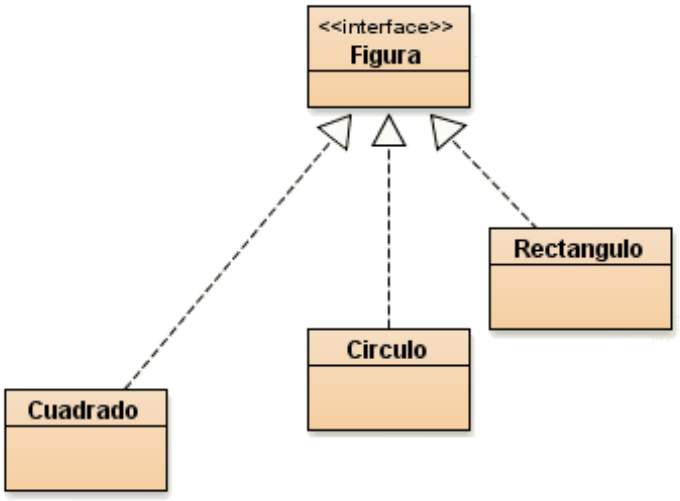
```
public class Circulo implements Figura {
      ...
}
```

6.1 Ejemplo 6

En este ejemplo vamos a crear una interfaz Figura y posteriormente implementarla en varias clases. En IntelliJ Idea para crear una nueva Interface seleccionamos crear una nueva clase, y seleccionamos Interface en el cuadro donde introducimos el nombre.



Vamos a ver un ejemplo simple de definición y uso de interfaz en Java. Las clases que vamos a usar y sus relaciones se muestran en el esquema:



```java
public interface Figura {
    float PI=3.1416f;
    float area();
}
```

```java
public class Rectangulo implements Figura {

    private float base;
    private float altura;

    public Rectangulo(float base, float altura){
        this.base=base;
        this.altura=altura;
    }
    @Override
    public float area() {
        return base*altura;
    }
}
```

```java
public class Cuadrado extends Rectangulo{

    public Cuadrado(float lado) {
        super(lado, lado);
    }
}
```

```java
public class Circulo implements Figura{

    private float diametro;

    public Circulo(float diametro){
        this.diametro=diametro;
    }
    @Override
    public float area() {
        return PI*diametro*diametro/4f;
    }
}
```

Un programa para probar las figuras podría ser

```
public class PruebaFiguras {
    public static void main(String[] args) {
        Figura cuad1=new Cuadrado(3.5f);
        Figura cuad2=new Cuadrado(2.2f);
        Figura cuad3=new Cuadrado(8.9f);

        Figura circ1=new Circulo(3.5f);
        Figura circ2=new Circulo(4f);

        Figura rect1=new Rectangulo(2.25f,2.55f);
        Figura rect2=new Rectangulo(12f,3f);

        ArrayList serieDeFiguras=new ArrayList();
        serieDeFiguras.add(cuad1);
        serieDeFiguras.add(cuad2);
        serieDeFiguras.add(cuad3);
        serieDeFiguras.add(circ1);
        serieDeFiguras.add(circ2);
        serieDeFiguras.add(rect1);
        serieDeFiguras.add(rect2);

        float areaTotal=0;
        for (Object unaFigura : serieDeFiguras) {
            Figura temporal=(Figura)unaFigura;
            areaTotal=areaTotal+temporal.area();
        }

        System.out.println("Tenemos un total de "+serieDeFiguras.size()+" figuras
con un área total de "+areaTotal);
    }
}
```

El resultado de ejecución sería:

```
/usr/lib/jvm/java-17/bin/java -javaagent:/home/jesua/IntelliJ idea
Tenemos un total de 7 figuras con un área total de 160.22504
```

En este ejemplo la interface Figura define un tipo de dato. Por ello podemos crear un ArrayList de figuras donde insertamos cuadrados, círculos, rectángulos, etc. (polimorfismo). Esto nos permite darle un tratamiento común a todas las figuras: Mediante un bucle while recorremos la lista de figuras y llamamos al método area() que será distinto para cada clase de figura (ligadura dinámica).